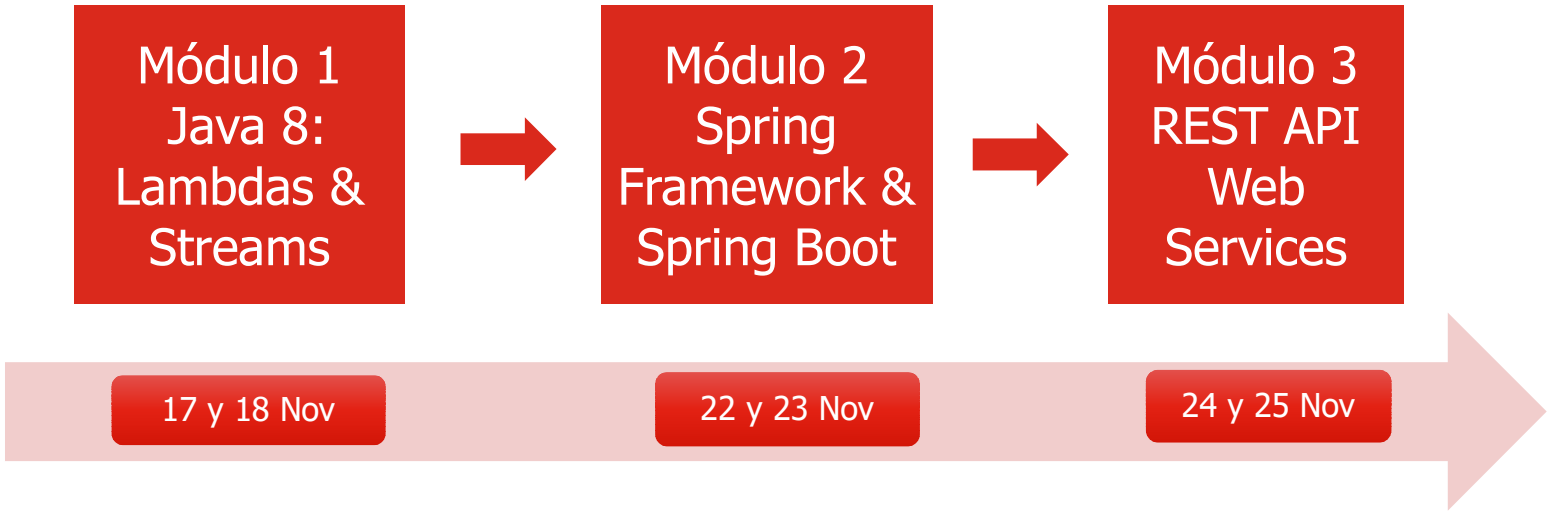
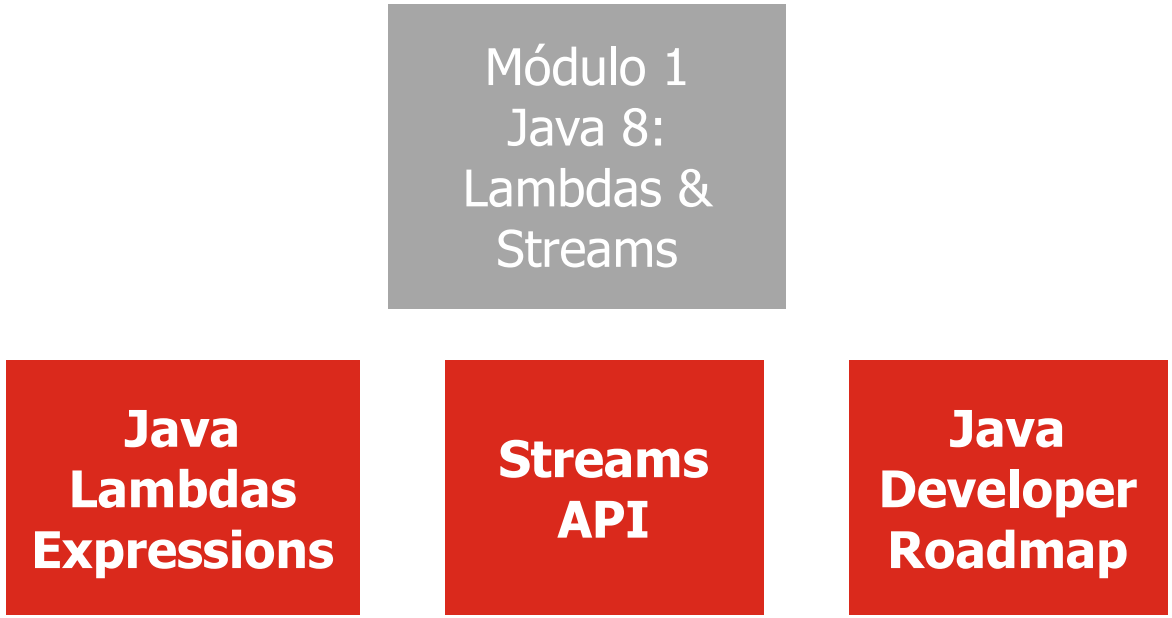
Programa Juniors Backend GFT



Formador: Ezequiel Llarena Borges

Bloques principales de contenidos



Objetivos del curso

Proporcionar una introducción a las expresiones Lambdas y Streams de Java 8:

- Utilizar expresiones Lambdas Java 8
- Utilizar Java 8 Streams API
- Tener una visión general del Java Developer Roadmap

Índice

1. Java SE 8

Evolución de JAVA

Sun Microsystems

Lanzamiento	Año
JDK Beta	1994
JDK 1.0	1996
JDK 1.1	1997
J2SE 1.2	1998
J2SE 1.3	2000
J2SE 1.4	2002
J2SE 5.0	2004
Java SE 6	2006

Evolución de JAVA

2010 - Sun Microsystem comprado por Oracle

Cambio principal introducido por Oracle:
- Plataforma abierta para Java: **Open JDK**
- Plataforma comercial uso empresarial: **Oracle Java**

Evolución de JAVA

Oracle Inc

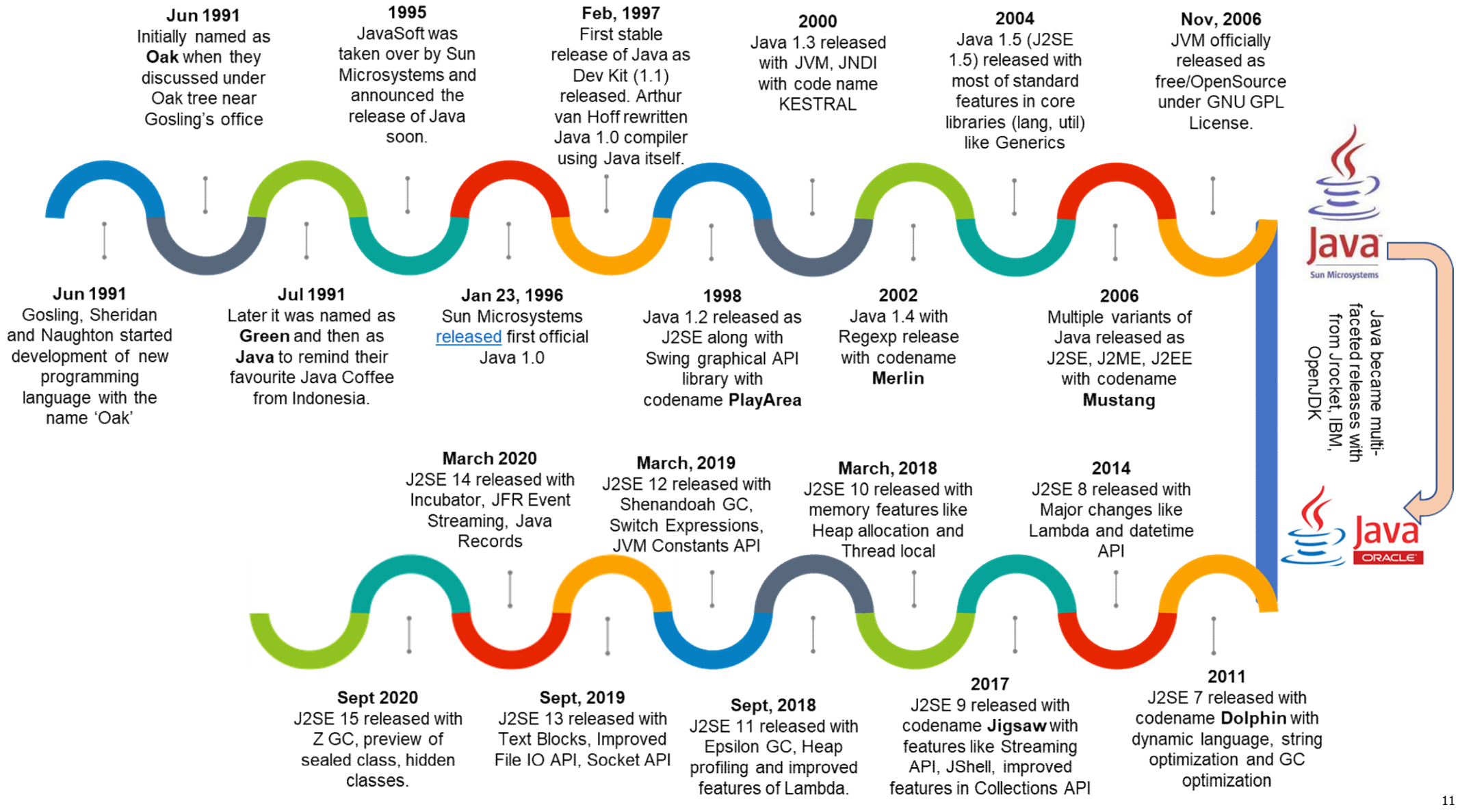
Lanzamiento	Año	
Java SE 7	2011	
Java SE 8	2014	LTS
Java SE 9	2017	
Java SE 10	2018	
Java SE 11	2018	LTS
Java SE 12	2019	

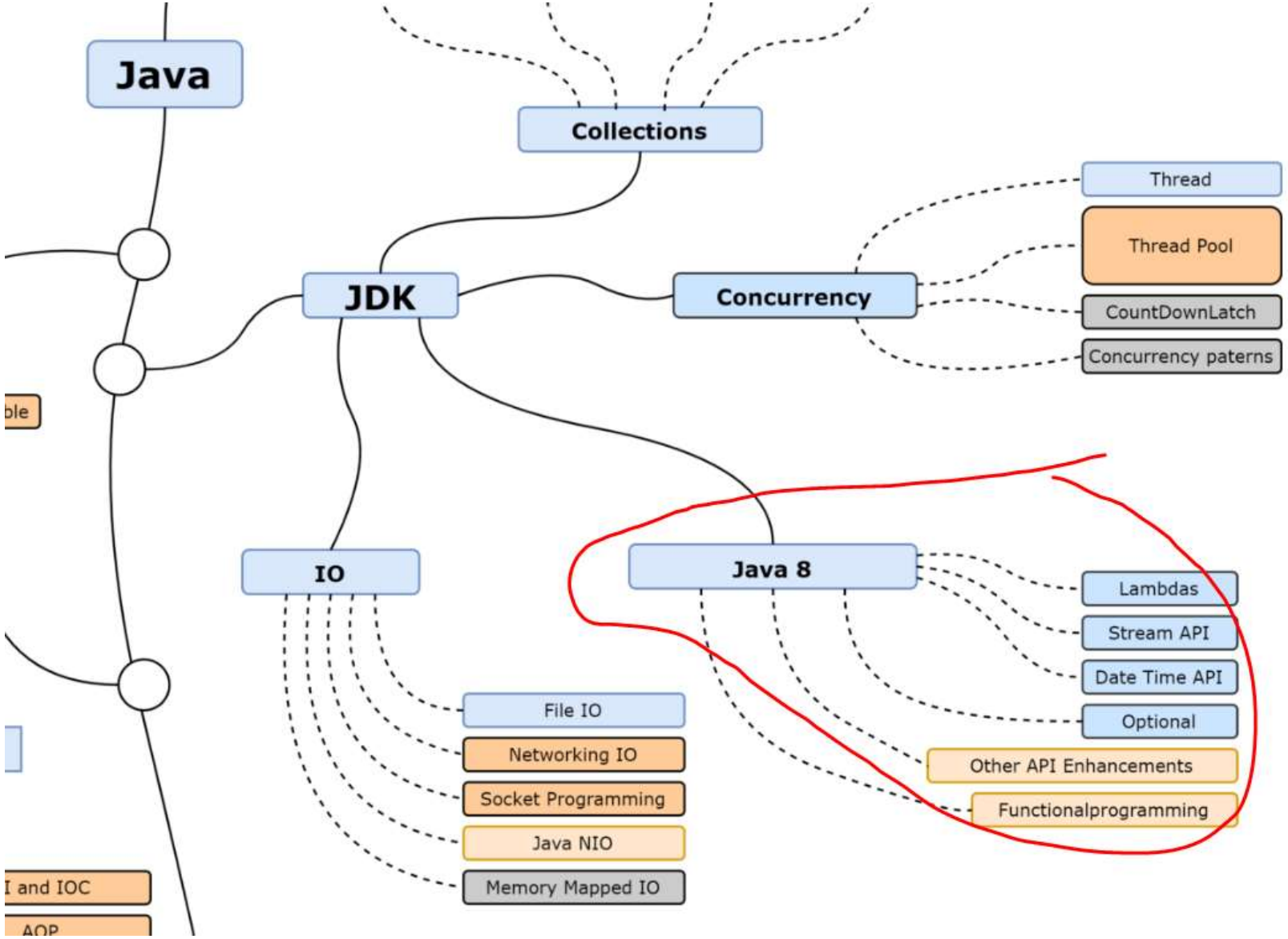
JAVA

Oracle JDK

Oracle JDK vs Open JDK

Comparativa	
Licencias	Oracle JDK las empresas o particulares no podrán tener acceso a los updates de la plataforma después de Enero de 2019 a menos que compres una licencia con Oracle. En el caso de Open JDK el uso es libre puedes crear una aplicación comercial o una sin animos de lucro.
Performance	No hay diferencia significativa entre ambas plataformas. La construcción de Oracle JDK está basada en Open JDK - En el caso de los bugs en Oracle JDK son corregidos inmediatamente y distribuidos a sus clientes de pago, en Open JDK debemos esperar a tener las correcciones (aportaciones de la comunidad Open Source).
Popularidad	Oracle JDK mantenida 100% por Oracle Corporation. Open JDK desarrollada por comunidad Java y una parte muy pequeña por Oracle y otras empresas como Red Hat, IBM, Azul systems, Apple Inc, SAP


Jun 1991
Initially named as Oak when they discussed under Oak tree near Gosling's office
1995
JavaSoft was taken over by Sun Microsystems and announced the release of Java soon.
Feb, 1997
First stable release of Java as Dev Kit (1.1) released. Arthur van Hoff rewritten Java 1.0 compiler using Java itself.
2000
Java 1.3 released with JVM, JNDI with code name KESTRAL
2004
Java 1.5 (J2SE 1.5) released with most of standard features in core libraries (lang, util) like Generics
Nov, 2006
JVM officially released as free/OpenSource under GNU GPL License.
Jun 1991
Gosling, Sheridan and Naughton started development of new programming language with the name 'Oak'
Jul 1991
Later it was named as Green and then as Java to remind their favourite Java Coffee from Indonesia.
Jan 23, 1996
Sun Microsystems released first official Java 1.0
1998
Java 1.2 released as J2SE along with Swing graphical API library with codename PlayArea
2002
Java 1.4 with Regexp release with codename Merlin
2006
Multiple variants of Java released as J2SE, J2ME, J2EE with codename Mustang
Java
Sun Microsystems
Java became multi-faceted releases with from Jrocket, IBM, OpenJDK
Java
ORACLE
March 2020
J2SE 14 released with Incubator, JFR Event Streaming, Java Records
March, 2019
J2SE 12 released with Shenandoah GC, Switch Expressions, JVM Constants API
March, 2018
J2SE 10 released with memory features like Heap allocation and Thread local
2014
J2SE 8 released with Major changes like Lambda and datetime API
Sept 2020
J2SE 15 released with Z GC, preview of sealed class, hidden classes.
Sept, 2019
J2SE 13 released with Text Blocks, Improved File IO API, Socket API
Sept, 2018
J2SE 11 released with Epsilon GC, Heap profiling and improved features of Lambda.
2017
J2SE 9 released with codename Jigsaw with features like Streaming API, JShell, improved features in Collections API
2011
J2SE 7 released with codename Dolphin with dynamic language, string optimization and GC optimization


Java
Collections
JDK
Concurrency
Thread
Thread Pool
CountDownLatch
Concurrency paterns
IO
Java 8
Lambdas
Stream API
Date Time API
Optional
Other API Enhancements
Functionalprogramming
File IO
Networking IO
Socket Programming
Java NIO
Memory Mapped IO

Java 8

Métodos default en interfaces

Declaración de comportamiento por defecto en método de interfaz

Interfaces funcionales

java.util.function
@FunctionalInterface

Lambda

Expresiones lambdas

Stream API

Permite realizar distintas operaciones funcionales sobre streams java.util.Stream

Date API

java.time API
tratamiento de fechas, tiempos, instantes y duraciones

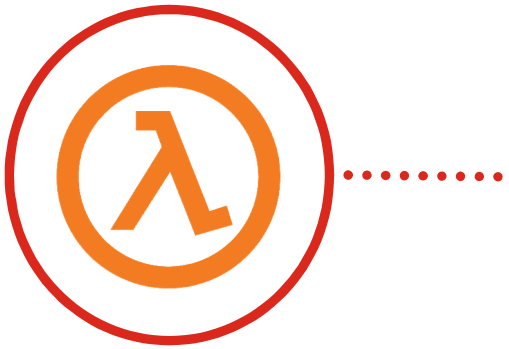
Excepciones y métodos nuevos en clases uso común

Integración con servicios Cloud

2. Lambda Expressions

Expresión Lambda de Java

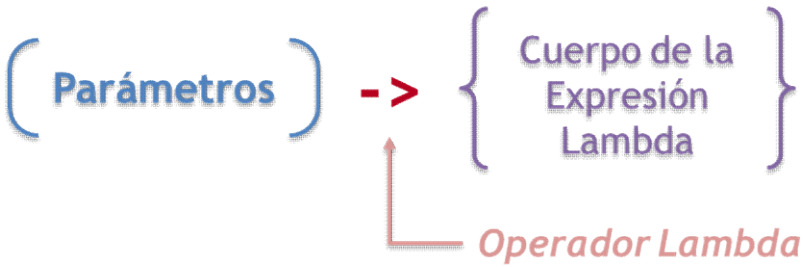
¿Qué es?



- **Función anónima**
- Método abstracto (declarado en una interfaz pero no implementado)
- Permite
  - tratar la funcionalidad o comportamiento como argumento de un método (el código como datos)
  - expresar instancias de clases de un solo método de forma más compacta
- Programación Imperativa vs Programación Funcional
- Soporte a la **programación funcional** junto con la API Stream

Expresión Lambda de Java

Sintaxis



- **(argumentos) -> { cuerpo }**
- Ejemplos:
  - (arg1, arg2,…) -> {cuerpo}
  - (int x) -> {cuerpo}
  - (int x, int y….) -> {cuerpo}
  - x -> {cuerpo}
  - () -> {cuerpo}

Expresión Lambda de Java

Sintaxis en los argumentos

Declaración Explícita de Argumentos	Argumentos Inferidos
(String str) -> {cuerpo}	(x) -> {cuerpo}
String str -> {cuerpo}	x -> {cuerpo}
(int a, long b) -> {cuerpo}	(a, b) -> {cuerpo}
() -> {cuerpo}	(a, b, int c) -> {cuerpo}
En ambos casos: si hay más de un argumento, entre paréntesis y separados por coma	

Expresión Lambda de Java

Sintaxis en el cuerpo

Entre llaves obligatorio	Opcional
Cuando devuelve más de un valor o el cuerpo tiene más de una instrucción	Cuando devuelve un sólo valor, aunque el compilador no muestra error si se ponen llaves
(int a, int b) -> { return a + b; }	() -> 10
() -> { return 3.1415; }	email -> System.out.println(email)
	s -> s.length() > 0

p -> {
return p.getGender() == Person.Sex.MALE
&& p.getAge() >= 18
&& p.getAge() <= 25;
}
	p -> p.getGender() == Person.Sex.MALE &&
	p.getAge() >= 18 && p.getAge() <= 25

Interfaz Funcional

¿Qué es?

- Interfaz que tiene **un solo método abstracto**
- Debe implementar sus métodos dentro la misma interfaz (esto no se podía hacer en versiones anteriores), para esto se debe anteponer la palabra reservada *default* al inicio de la declaración del método.
- @FunctionalInterface (opcional)

Interfaz Funcional

Ejemplo

```
@FunctionalInterface
public interface IFuncionLambda {
   public void calcular(int a, int b);
}
```

```
public class TestLambdas {
    public static void main(String[] args) {

        int x = 10;
        int y = 5;

        IFuncionLambda iflambda =

           (a, b) -> { System.out.println(a + b);   };

        iflambda.calcular(x, y);
    }
}
```

Expresión Lambda de Java

Tipos

Predicados	Funciones	Proveedores	Consumidores
Predicate<T>	Function<T,R>	Supplier<T>	Consumer<T>
expresiones que reciben un argumento y devuelven un valor lógico	reciben un argumento y devuelven un resultado	no tiene parámetros de entrada, pero si devuelven un resultado	un sólo argumento de entrada y no devuelven ningún valor

3. Stream API

API Stream

¿Qué es?

- Biblioteca de clases para realizar operaciones con **streams** de Java
- API Stream permite escribir código eficiente, limpio y conciso

stream()
- Un array es una colección de datos. Por lo que es capaz de crear un stream
- Proporciona un stream secuencial
- métodos sobrecargado para diferentes tipos
  - double, int, long, Object

Stream

Descripción

Operaciones intermedias

Fuente

```
int total = transactions.stream()
    .filter(t -> t.getBuyer().getCity().equals("London"))
    .mapToInt(Transaction::getPrice)
    .sum();
```

Operación terminal

Stream Interface

Intermediate Operations

Un Stream proporciona una secuencia de elementos

- Soportan operaciones de agregado secuenciales o en paralelo

La mayoría de la operaciones requieren un parámetro que describe su comportamiento

- En general utilizando expresiones lambda
- La mayoría no modifican el stream (Non-Interfering)
- En gneral son sin estado

Stream Interface

Operaciones Intermedias: filtrado y mapeo

distinct()
- Regresa un flujo sin elementos duplicados

filter(Predicado p)
- Regresa un flujo con aquellos elementos que cumplen el predicado

map(Function p)
- Regresa un stream en donde la función proporcionada es aplicada a cada uno de los elementos del stream

mapToInt(), mapToDouble(), mapToLong()
- Funcionan como map() pero producen streams de primitivas es vez de objetos

Stream Interface

Operaciones Terminales

Finaliza la tubería de operaciones sobre el stream

Unicamente haste este punto se realizan los procesamientos

- Esto permite la optimización de la tubería
- Evaluación perezosa (Lazy)
- Operaciones de fusionado o de fusionado
- Eliminación de operaciones redundantes
- Ejecución en paralelo

Genera un resultado explícito o un efecto secundario

Stream Interface

Operaciones Terminales: elementos coincidentes

findFirst()
- La primer coincidencia

findAny()
- Trabaja igual que findFirst(), pero actúa sobre un stream paralelo
- boolean allMatch(Predicate p)
  - Si todos los elementos del stream coinciden utilizando el Predicado p
- boolean anyMatch(Predicate p)
  - Si al menos uno de los elmeentos coincide con la condición del predicado p
- boolean noneMatch(Predicate p)
  - ninguno de los elementos coincide de acuerdo al predicado

Stream Interface

Operaciones Terminales: resultados numéricos

count()
- Regresa el numero de elementos que hay en el stream

max (Comparator c)
- El elemento con el máximo valor que está dentro del stream en base al comparador

min(Comparador c)
- El elemento con el valor más pequeño que está dentro del stream en base al comparador
- Regresa un Optional, si el stream esta vacío

average()
- Regresa la media aritmética del stream
- Si el stream está vacío. regresa un Optional

sum()
- Regresa la suma de los elementos del stream

Stream Interface

Operaciones Terminales: iteración

forEach(Consumer c)

- Realiza una acción por cada elemento del stream

forEachOrdered(Consumer c)

- Funciona igual que forEach, pero asegura que el orden de los elementos (Si hay elementos) es respetado cuando se utiliza por un styream paralelo

Utilicelo con precaución

- Alienta el estilo de programación no-funcional (imperativo)

Stream Interface

Clase Optional

- Ciertas situaciones en Java devuelven como resultado un valor null: referencia a un objeto que no ha sido inicializado

Las operaciones terminales como min() , max() podrían no poder regresar un resultado directo

- Suponga que el stream de entrada esta vacío

Optional <T>

- Contenedor para una referencia de objeto (null o objeto real)
- Considerelo como un stream con 0 o 1 elemento
- Garantizado que la referencia Optional regresada no es null

Stream

Referencia a métodos (Java 8)

- Permite sustituir expresiones lambda por referencias a los métodos del objeto utilizando el operador ***::***
- ***Ejemplo:***

```
public class TestReferenciaMetodos {

    public static void main(String[] args) {
        List names = new ArrayList();
          names.add("Andrea");
          names.add("Luisa");
          names.add("Diego");
          names.add("Paúl");
          names.add("Dario");
          names.forEach(System.out::println);
    }

}
```

Stream

Referencia a métodos (Java 8)

```java
public int calcularPrecioTotalLambda() {
    int precioTotal = this.precios.stream().mapToInt(precio -> precio.intValue()).sum();
    return precioTotal;
}
```

- Las dos formas hacen lo mismo: obtener de cada Integer su valor. En el primer caso (i -> i.intValue()) llamamos al método intValue de cada Integer.
- En el segundo caso (Integer::intValue) hacemos uso del método a través de una referencia

```java
public int calcularPrecioTotalLambda() {
    int precioTotal = this.precios.stream()
                        .mapToInt(Integer::intValue)
                        .sum();
    return precioTotal;
}
```

Stream

Referencia a métodos (Java 8)

- Implementación de la expresión Lambda (tipo proveedor) utilizando la interfaz Supplier:

```
import java.util.function.Supplier;

public class TestLambda {

    public static void main(String[] args) {
            //se crea un proveedor de tipo Persona, el cual obtiene una persona
        Supplier<Persona> supplier = TestLambda::llenarPersona;
            //obtiene desde el proveedor la persona y la asigna a per
        Persona per = supplier.get();
            // imprime el nombre
        System.out.println(per.getNombre());
    }
    // asigna los nombres y dirección a la persona
    public static Persona llenarPersona(){
        return new Persona("Pablo", "Andrade", "Loja");
    }
}
```

```
public class Persona {
    private String nombre;
    private String apellido;
    private String direccion;
```

Stream

Parallel stream

- Los streams se pueden ejecutar en serie o en paralelo. Cuando un stream se ejecuta en paralelo, el tiempo de ejecución de Java divide el flujo en múltiples substreams. Las operaciones de agregación iteran y procesan estos substreams en paralelo y luego combinan los resultados.

```java
public boolean detectarErrorAnyMatchParallel() {
    return this.precios.parallelStream().anyMatch(precio -> precio.intValue() < 0);
}

public boolean detectarErrorFindAnyParallel() {
    return this.precios.parallelStream().filter(precio -> precio.intValue() < 0)
                                        .findAny()
                                        .isPresent();
}

public boolean detectarErrorFindFirstParallel() {

    return this.precios.parallelStream().filter(precio -> precio.intValue() < 0)
                                        .findFirst()
                                        .isPresent();
}
```

4. Date API

API Date

Java 8 Date/Time API

- Se basa en la biblioteca de Java Joda Time
- API muy clara, concisa y fácil de entender
- Simplifica drásticamente el procesamiento de la fecha y la hora y corrige muchas deficiencias de la antigua biblioteca de fechas
- Flexibilidad: múltiples representaciones del tiempo.: java.util.time.*
- Immutability and Thread-Safety
  - Todas las representaciones de tiempo en la API de fecha y hora de Java 8 son inmutables y, por tanto, seguras para los hilos.
  - Todos los métodos de mutación devuelven una nueva copia en lugar de modificar el estado del objeto original.
  - Las antiguas clases como java.util.Date no eran seguras para los hilos y podían introducir errores de concurrencia muy sutiles.

API Date

Java 8 Date/Time API

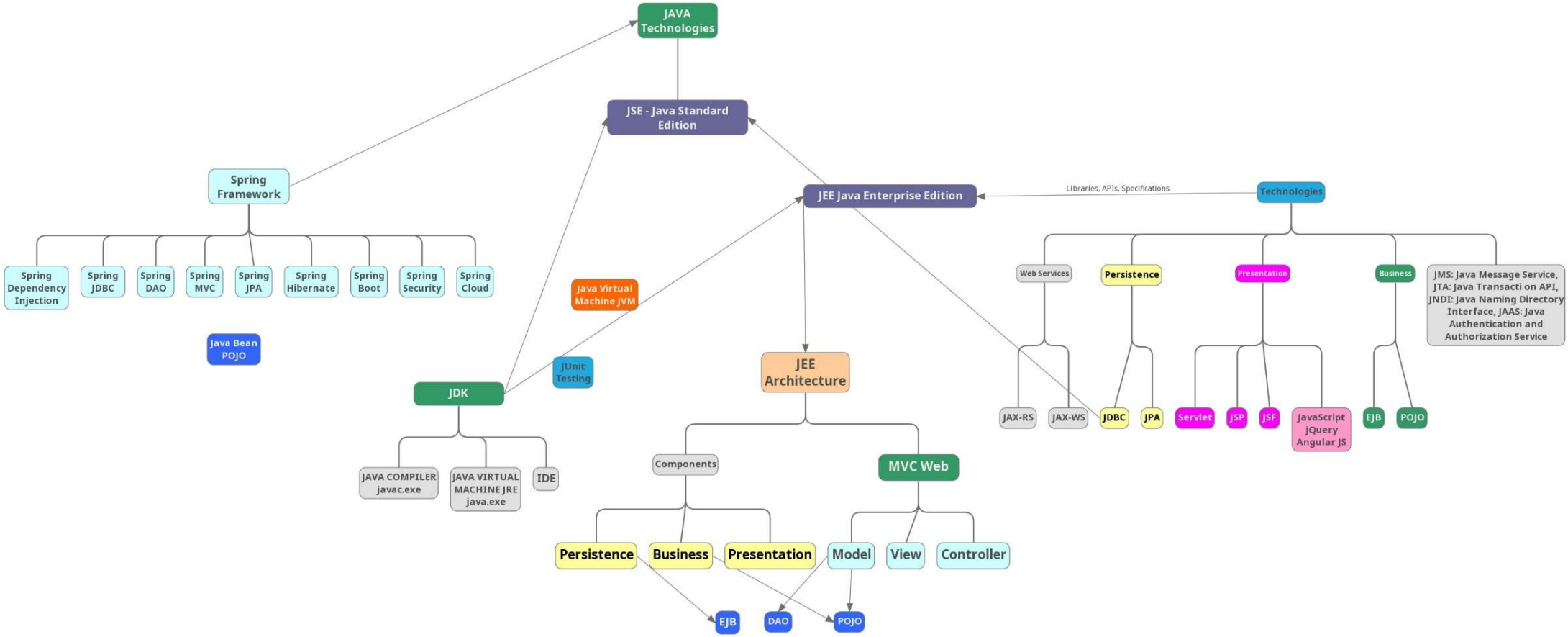
- Flexibilidad: múltiples representaciones del tiempo.: **java.util.time.***

  - *Instant* – represents a point in time (timestamp)
  - *LocalDate* – represents a date (year, month, day)
  - *LocalDateTime* – same as *LocalDate*, but includes time with nanosecond precision
  - *OffsetDateTime* – same as *LocalDateTime*, but with time zone offset
  - *LocalTime* – time with nanosecond precision and without date information
  - *ZonedDateTime* – same as *OffsetDateTime*, but includes a time zone ID
  - *OffsetLocalTime* – same as *LocalTime*, but with time zone offset
  - *MonthDay* – month and day, without year or time
  - *YearMonth* – month and year, without day or time
  - *Duration* – amount of time represented in seconds, minutes and hours. Has nanosecond precision
  - *Period* – amount of time represented in days, months and years

API Date

	Old API	New API
Getting current time	Date now = new Date();	ZonedDateTime now = ZonedDateTime.now();
Representing specific time	Date birthDay = new GregorianCalendar(1990, Calendar.DECEMBER, 15).getTime();	LocalDate birthDay = LocalDate.of(1990, Month.DECEMBER, 15);
Extracting specific fields	int month = new GregorianCalendar().get(Calendar.MONTH);	Adding and subtracting timeMonth month = LocalDateTime.now().getMonth();
Adding and subtracting time	GregorianCalendar calendar = new GregorianCalendar(); calendar.add(Calendar.HOUR_OF_DAY, -5); Date fiveHoursBefore = calendar.getTime();	LocalDateTime fiveHoursBefore = LocalDateTime.now().minusHours(5);
Number of days in a month	Calendar calendar = new GregorianCalendar(1990, Calendar.FEBRUARY, 20); int daysInMonth = calendar.getActualMaximum(Calendar.DAY_OF_MONTH);	int daysInMonth = YearMonth.of(1990, 2).lengthOfMonth();

5. Java Developer Roadmap


JAVA Technologies
JSE - Java Standard Edition
Spring Framework
JEE Java Enterprise Edition
Libraries, APIs, Specifications
Technologies
Spring Dependency Injection
Spring JDBC
Spring DAO
Spring MVC
Spring JPA
Spring Hibernate
Spring Boot
Spring Security
Spring Cloud
Java Bean POJO
Java Virtual Machine JVM
JUnit Testing
JDK
JEE Architecture
Web Services
Persistence
Presentation
Business
JMS: Java Message Service, JTA: Java Transacti on API, JNDI: Java Naming Directory Interface, JAAS: Java Authentication and Authorization Service
JAX-RS
JAX-WS
JDBC
JPA
Servlet
JSP
JSF
JavaScript jQuery Angular JS
EJB
POJO
JAVA COMPILER javac.exe
JAVA VIRTUAL MACHINE JRE java.exe
IDE
Components
MVC Web
Persistence
Business
Presentation
Model
View
Controller
EJB
DAO
POJO

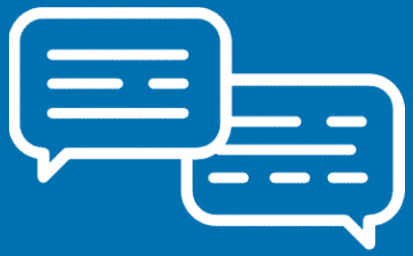
Bibliografía y webgrafía

Webs de referencia

https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

http://stackoverflow.com/

http://www.codecademy.com/



DEBATE

Programación Funcional

OBJECTIVE

Relacionar y consolidar los conceptos de las expresiones lambda, Stream y la programación funcional.

INSTRUCTIONS

1. Debatir en grupo qué entiendes por Programación Funcional
2. Poner en común todas las aportaciones y sacar conclusiones entre todos.

15 min

REFLEXIONES

15 min